1. **What is the new features introduced in Java 5,6,7,8?**

   Java 5: For-each loop (Java 5), Varargs (Java 5), Static Import (Java 5), Autoboxing and Unboxing (Java 5), Enum (Java 5), Covariant Return Type (Java 5), Annotation (Java 5), Generics (Java 5)

   Java 6: JDBC 4, @Override Annotation

   Java 7: String in Switch, Try with resource, catching multiple exceptions under single catch

   Java 8: Lambda Expressions, Functional Interface, Static and Default Methods, Optional, Method Reference, Date and Time API, Streams.

2. **What is the use of covariant?**

   Before JDK 5.0, it was not possible to override a method by changing the return type. When we override a parent class method, the name, argument types and return type of the overriding method in child class has to be exactly same as that of parent class method. Overriding method was said to be invariant with respect to return type.

   Java 5.0 onwards it is possible to have different return type for an overriding method in child class, but child's return type should be sub-type of parent's return type. Covariant return type works only for non-primitive return types.

```
class SuperClass {
   SuperClass get() {
      System.out.println("SuperClass");
      return this;
   }
}
public class Tester extends SuperClass {
   Tester get() {
      System.out.println("SubClass");
      return this;
   }
   public static void main(String[] args) {
      SuperClass tester = new Tester();
      tester.get();
   }
}
```

Output

```
Subclass
```

3. **How to increase the heap size?**

Here is how you can increase the heap size using JVM using command line:

```
-Xms<size>        set initial Java heap size
-Xmx<size>        set maximum Java heap size
-Xss<size>        set java thread stack size

java -Xms16m -Xmx64m ClassName
```

In the above line we can set minimum heap to 16mb and maximum heap 64mb

4. **Difference between Class.forName() and ClassLoader.loadClass()?**

   Class.forName(): load and initialize the class. In class loader subsystem it executes all the three phases i.e. load, link, and initialize phases.
   ClassLoader.loadClass(): behavior, which delays initialization until the class is used for the first time. In class loader subsystem it executes only two phases i.e. load and link phases.

5. **Explain about ThreadLocal?**

   ThreadLocal in Java is another way to achieve thread-safety. In thread local, you can set any object and this object will be local and global to the specific thread which is accessing this object.

   Java ThreadLocal class provides thread-local variables. It enables you to create variables that can only be read and write by the same thread. If two threads are executing the same code and that code has a reference to a ThreadLocal variable then the two threads can't see the local variable of each other.

   As part of ThreadLocal class we have three methods,

   - ✓ Set
   - ✓ Get
   - ✓ Remove

```
ThreadLocal<Number> gfg_local = new ThreadLocal<Number>();

ThreadLocal<String> gfg = new ThreadLocal<String>();
// setting the value
gfg_local.set(100);

// returns the current thread's value
System.out.println("value = " + gfg_local.get());

// setting the value
gfg_local.set(90);

// returns the current thread's value of
System.out.println("value = " + gfg_local.get());

// setting the value
gfg_local.set(88.45);

// returns the current thread's value of
System.out.println("value = " + gfg_local.get());

// setting the value
gfg.set("GeeksforGeeks");

// returns the current thread's value of
System.out.println("value = " + gfg.get());
```

Output:

```
value = 100
value = 90
value = 88.45
value = GeeksforGeeks
```

6. **Explain abut Thread Scheduler?**

Thread scheduler in java is the part of the JVM that decides which thread should run. There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

7. **Difference between @RequestParam, @QueryParam, @PathParam and @PathVariable?**

@RequestParam annotation used for accessing the query parameter values from the request. @RequestParam is more useful on a traditional web application where data is mostly passed in the query parameters. @RequestParam annotation can specify default values if a query parameter is not present or empty by using a defaultValue attribute, provided the required attribute is false.

```
http://localhost:8080/springmvc/hello/101?param1=10&param2=20
```

n the above URL request, the values for param1 and param2 can be accessed as below:

```
public String getDetails(
    @RequestParam(value="param1", required=true) String param1,
        @RequestParam(value="param2", required=false) String param2){
...
}
```

The following are the list of parameters supported by the @RequestParam annotation:

- **defaultValue** – This is the default value as a fallback mechanism if request is not having the value or it is empty.
- **name** – Name of the parameter to bind
- **required** – Whether the parameter is mandatory or not. If it is true, failing to send that parameter will fail.
- **value** – This is an alias for the name attribute

@PathVariable: Identifies the pattern that is used in the URI for the incoming request. It is best suitable in Spring MVC application. We can use it in Rest service also.

http://localhost:8080/springmvc/hello/101?param1=10&param2=20

```
@RequestMapping("/hello/{id}")    public String getDetails(@PathVariable(value="id") String id,
    @RequestParam(value="param1", required=true) String param1,
    @RequestParam(value="param2", required=false) String param2){
.......
}
```

@QueryParam: Request parameters in query string can be accessed using @QueryParam annotation.

```
@GET
@Produces("application/json")
@Path("json/companyList")
public CompanyList getJSON(@QueryParam("start") int start, @QueryParam("limit") int limit) {
  CompanyList list = new CompanyList(companyService.listCompanies(start, limit));
  return list;
}
```

@PathParam: It will injects value from URI to your method input parameters.

http://localhost:8080/books/1234

```
@Path("/library")
public class Library {

   @GET
   @Path("/book/{isbn}")
   public String getBook(@PathParam("isbn") String id) {
      // search my database and get a string representation and return it
   }
}
```

8. **Explain about idempotent methods?**

Idempotent operations produce the same result even when the operation is repeated many times. The result of the 2nd, 3rd, and 1,000th repeat of the operation will return exactly the same result as the 1st time.

Below are the idempotent methods

- ✓ Get
- ✓ Put
- ✓ Head
- ✓ Delete

9. **Benefits of encapsulation?**

- ✓ A class can have total control over what is stored in its fields.
- ✓ Data Hiding
- ✓ Reusability
- ✓ Easy to test

10. **Abstraction vs Encapsulation?**

Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It focuses on the **external** lookout.	It focuses on **internal** working.
It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public protected).
It is the process of **gaining** information.	It is the process of **containing** the information.
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.

11. **How to convert from HTTP to HTTPS?**

- ✓ Generate or buy the SSL certificate. We can generate the self-certificate by using keytool command,

```
keytool -genkeypair -alias tomcat -keyalg RSA -keysize 2048 -keystore keystore.jks -validity 3650 -storepass password
```

- ✓ After certificate generation add all the certificate details in application.properties

```
server.port=8443

server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=password
server.ssl.key-alias=tomcat

security.require-ssl=true
```

- ✓ Block the HTTP request by using Spring Security

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .requiresChannel()
            .anyRequest()
            .requiresSecure();
    }
}
```

- ✓ If required add the code to redirect HTTP request to HTTPS.

12. Difference java.util.date and SQL date?

SQL Date just represent DATE without time information while java.util.Date represents both Date and Time information.

13. Can we have equals without hashcode?

Yes, you can only implement equals() method without implementing hashcode() method.

But standard practice says that you should implement both of them and for the equal object the hashcode should be the same.

14. Can we have hashcode without equals?

Yes, you can only implement hashcode() method without implementing equals() method.

But standard practice says that you should implement both of them and for the equal object the hashcode should be the same.

15. **Why we need to override the equals and hashcode?**

HashMap, HashTable and HashSet use the hashcode value of an object to find out how the object would be stored in the collection, and subsequently hashcode is used to help locate the object in the collection. Hashing retrieval involves:

- ✓ First, find out the right bucket using hashCode().
- ✓ Secondly, search the bucket for the right element using equals()

16. **Does ArrayList used hashcode() and equals()?**

The hashCode of ArrayList is a function of the hashCode of all the elements stored in the ArrayList, so it doesn't change when the capacity changes, it changes whenever you add or remove an element or mutate one of the elements in a way that changes its hashCode.

17. **Is it possible to have Hashmap without hashcode and equals?**

If you are maintaining the key as object then it is possible. Any java object that does not define its own equals and hashcode methods inherit the default equals and hashcode methods on java.lang.Object. These default implementation are based on object reference equality and not on logically equality. Since you have called get with the same object reference the object can be returned from the map.

18. **HashSet internally uses HashMap but HashSet does not store key value pair, In internally how its handled?**

Actually the value we insert in HashSet acts as a key to the map Object and for its value, java uses a constant variable. So in key-value pair, all the values will be the same.

```
// Dummy value to associate with an Object in Map
private static final Object PRESENT = new Object();
```

If we look at the ***add()*** method of HashSet class:

```
public boolean add(E e)
{
   return map.put(e, PRESENT) == null;
}
```

We can notice that, add() method of HashSet class internally calls the ***put()*** method of backing the HashMap object by passing the element you have specified as a key and constant "PRESENT" as its value. ***remove()*** method also works in the same manner. It internally calls remove method of Map interface.

19. Can we make Restful web services stateful?

Yes. Rest engages in state transfer and to make them stateful, we can use client side or db persisted session state, and transfer them across web service invocations as an attribute in either the header or a method parameter.

20. Explain the techniques to tune the database?

- ✓ Apply the Normal Forms
- ✓ While using SELECT statement, only fetch whatever information is required and avoid using * in your SELECT queries because it would load the system unnecessarily.
- ✓ Create your indexes carefully on all the tables where you have frequent search operations. Avoid index on the tables where you have less number of search operations and more number of insert and update operations.
- ✓ For queries that are executed on a regular basis, try to use procedures. A procedure is a potentially large group of SQL statements.
- ✓ Avoid the loops.

21. Explain about Swagger?

A Swagger is an open-source tool. It is the most popular API documentation format for RESTful Web Services. It provides both JSON and UI support. JSON can be used as a machine-readable format, and Swagger-UI is for visual display.

Currently we are using Swagger 2. Latest version of swagger is 2.2.1

To enable Swagger we need to add the dependent jars and Swagger configuration class,

```
@Configuration
//Enable Swagger
@EnableSwagger2
public class SwaggerConfig
{
//creating bean
@Bean
public Docket api()
{
//creating constructor of Docket class that accepts parameter DocumentationType
return new Docket(DocumentationType.SWAGGER_2);
}
}
```

JSON format Documentation: http://localhost:8080/v2/api-docs

Swagger UI: http://localhost:8080/swagger-ui.html

Swagger Annotations:

@ApiModel: It provides additional information about Swagger Models.
@ApiModelProperty: It allows controlling swagger-specific definitions such as values, and additional notes.

```
@ApiModel(description="All details about the user")
public class User
{
private Integer id;
@Size(min=5, message="Name should have atleast 5 characters")
@ApiModelProperty(notes="name should have atleast 5 characters")
private String name;
```

@Api – We can add this Annotation to the controller to add basic information regarding the controller.

```
@Api(value = "Swagger2DemoRestController", description = "REST APIs related to Student Entity!!!!")
@RestController
public class Swagger2DemoRestController {
    ...
}
```

@ApiOperation and @ApiResponses – We can add these annotations to any rest method in the controller to add basic information related to that method.

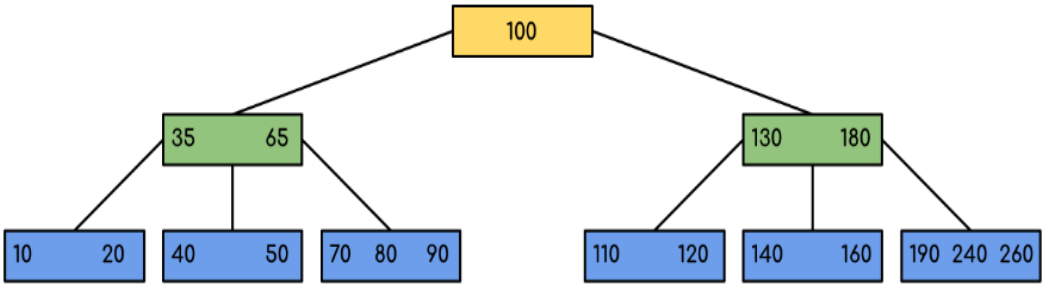
```
@ApiOperation(value = "Get list of Students in the System ", response = Iterable.class, tags = "getStudents")
@ApiResponses(value = {
            @ApiResponse(code = 200, message = "Success|OK"),
            @ApiResponse(code = 401, message = "not authorized!"),
            @ApiResponse(code = 403, message = "forbidden!!!"),
            @ApiResponse(code = 404, message = "not found!!!") })

@RequestMapping(value = "/getStudents")
public List<Student> getStudents() {
    return students;
}
```

22. How HashMap performance increased in java8?

Hash collisions have negative impact on the lookup time of HashMap. When multiple keys end up in the same bucket, then values along with their keys are placed in a linked list. In case of retrieval, linked list has to be traversed to get the entry. In worst case scenario, when all keys are mapped to the same bucket, the lookup time of HashMap increases.

- ✓ To address this issue, Java 8 hash elements **use balanced trees (B Tree)** instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing

Entry objects in a linked list but after the number of items in a hash becomes larger than a certain threshold. The hash will change from using a linked list to a balanced tree.



- ✓ The alternative String hash function added in Java 7 has been removed.

23. Give the best places for Immutable classes?

- ✓ Multithreading environment
- ✓ HashMap Key

24. What is meant by cloning and explain different types of cloning?

Cloning is the process of getting the exact copy of an object. The clone() method of Object class is used to clone an object.

The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

Below are the types of cloning,

Shallow Cloning: A shallow copy of an object copies the 'main' object, but doesn't copy the inner objects. If we make changes in shallow copy then changes will get reflected in the source object. Both instances are not independent.

```
class Department {
    String empId;
    public Department(String empId) {
        this.empId = empId;
    }
}
 class Employee implements Cloneable {
    int id;
    Department dept;

    public Employee(int id,Department dept) {
        this.id = id;
        this.dept = dept;
    }
     // Default version of clone() method. It creates shallow copy of an object.
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
 public class ShallowCopyInJava {
     public static void main(String[] args) {

        Department dept1 = new Department ("1", "A", "AVP");
        Employee emp1 = new Employee (111, "John", dept1);
        Employee emp2 = null;

        try {
            // Creating a clone of emp1 and assigning it to emp2
            emp2 = (Employee) emp1.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
         // Printing the designation of 'emp1'

        System.out.println(emp1.dept.designation); // Output : AVP

        // Changing the designation of 'emp2'
        emp2.dept.designation = "Director";

        // This change will be reflected in original Employee 'emp1'
        System.out.println(emp1.dept.designation); // Output : Director
    }
```

Deep Cloning: A Deep copy of an object copies the ‘main’ object and its inner objects also. If we make changes in deep copy then changes will not reflected in the source object. Both instances are independent.

```
class Department {
    String empId;
    public Department(String empId) {
        this.empId = empId;
    }
}
 class Employee implements Cloneable {
    int id;
    Department dept;

    public Employee(int id,Department dept) {
        this.id = id;
        this.dept = dept;
    }
     // Default version of clone() method. It creates shallow copy of an object.
    protected Object clone() throws CloneNotSupportedException {
        Employee emp = (Employee) super.clone();
        emp.dept = (Department) dept.clone();
        return emp;
    }
}
 public class ShallowCopyInJava {
     public static void main(String[] args) {
        Department dept1 = new Department("1", "A", "AVP");
        Employee emp1 = new Employee(111, "John", dept1);
        Employee emp2 = null;
        try {
            // Creating a clone of emp1 and assigning it to emp2
            emp2 = (Employee) emp1.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        // Printing the designation of 'emp1'
        System.out.println(emp1.dept.designation); // Output : AVP

        // Changing the designation of 'emp2'
        emp2.dept.designation = "Director";

        // This change will be reflected in original Employee 'emp1'
        System.out.println(emp1.dept.designation); // Output : AVP
    }
}
```

25. How to convert Monolith to Micro Services?

- **Split the Front end and Backend**
  - ✓ Presentation Layer
  - ✓ Business Layer
  - ✓ Data-access Layer
- **Analyze the database and make it independent tables.**

26. How to avoid the Dead Lock?

The deadlock is a situation when two or more threads try to access the same object that is acquired by another thread. Since the threads wait for releasing the object, the condition is known as deadlock.

We can avoid the dead lock by using below techniques,

- ✓ Use Thread.join() method
- ✓ Apply the timeout on each thread
- ✓ Apply the Thread priorities

27. Difference between Class level and Object level locking?

Object Level Locks – It can be used when you want non-static method or non-static block of the code should be accessed by only one thread.

Class Level locks – It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime. It should always be used to make static data thread safe.

Example of Class Level Lock

```
public class ClassLevelLockExample {
   public void classLevelLockMethod() {
      synchronized (ClassLevelLockExample.class) {
         //DO your stuff here
      }
   }
}
```

Example of Object Level Lock

```
public class ObjectLevelLockExample {
   public void objectLevelLockMethod() {
      synchronized (this) {
         //DO your stuff here
      }
   }
}
```

28. What is the use of @Primary annotation?

@Primary to give higher preference to a bean when there are multiple beans of the same type.

In some cases, we need to register more than one bean of the same type.

```
@Configuration
public class Config {

    @Bean
    public Employee JohnEmployee() {
        return new Employee("John");
    }

    @Bean
    public Employee TonyEmployee() {
        return new Employee("Tony");
    }
}
```

To access beans with the same type we usually use @Qualifier(“beanName”) annotation. We apply it at the injection point along with @Autowired. In our case, we select the beans at the configuration phase so @Qualifier can't be applied here.

```
@Configuration
public class Config {

    @Bean
    public Employee JohnEmployee() {
        return new Employee("John");
    }

    @Bean
    @Primary
    public Employee TonyEmployee() {
        return new Employee("Tony");
    }
}
```